Project 1\_5

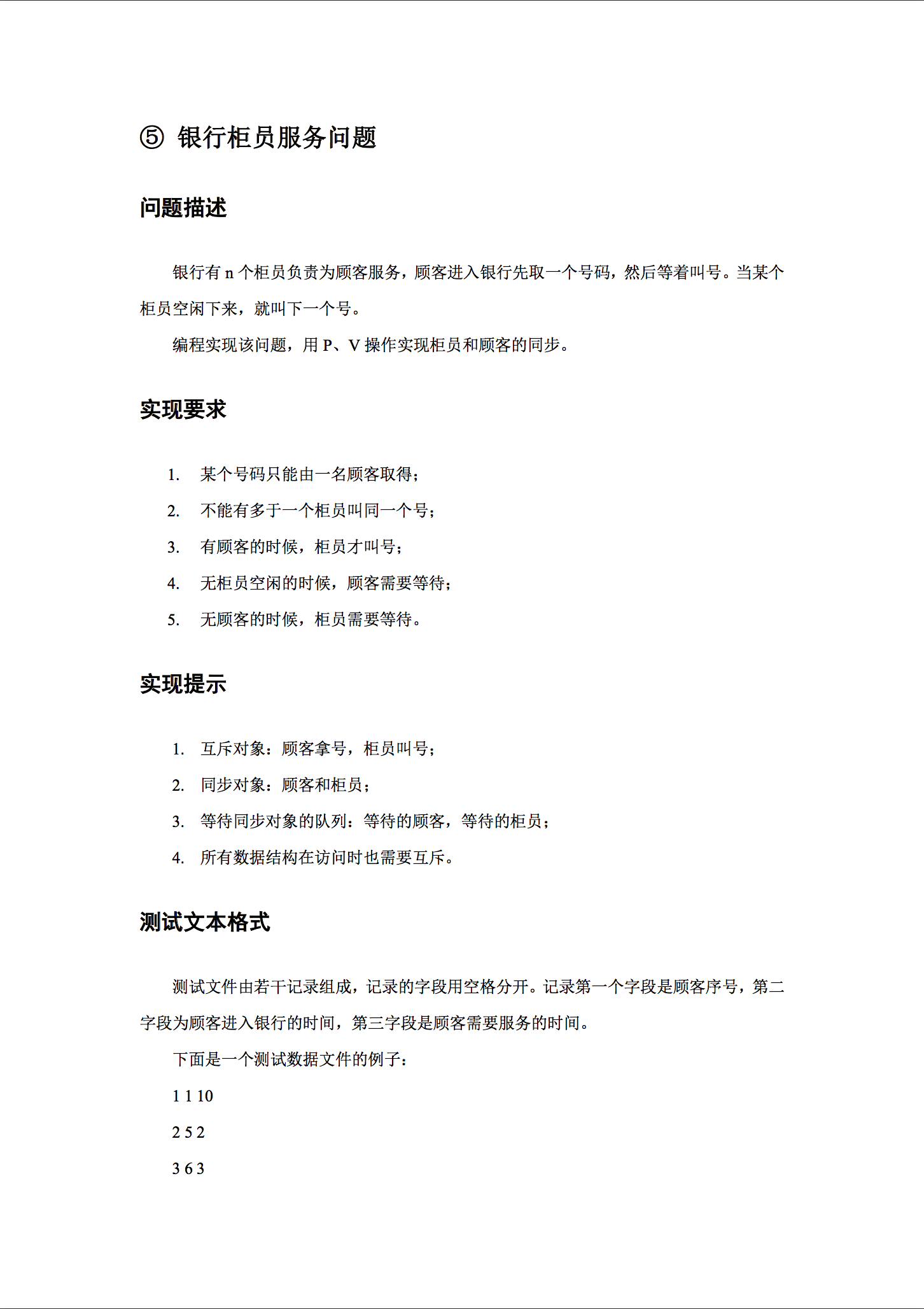
银行柜员服务问题

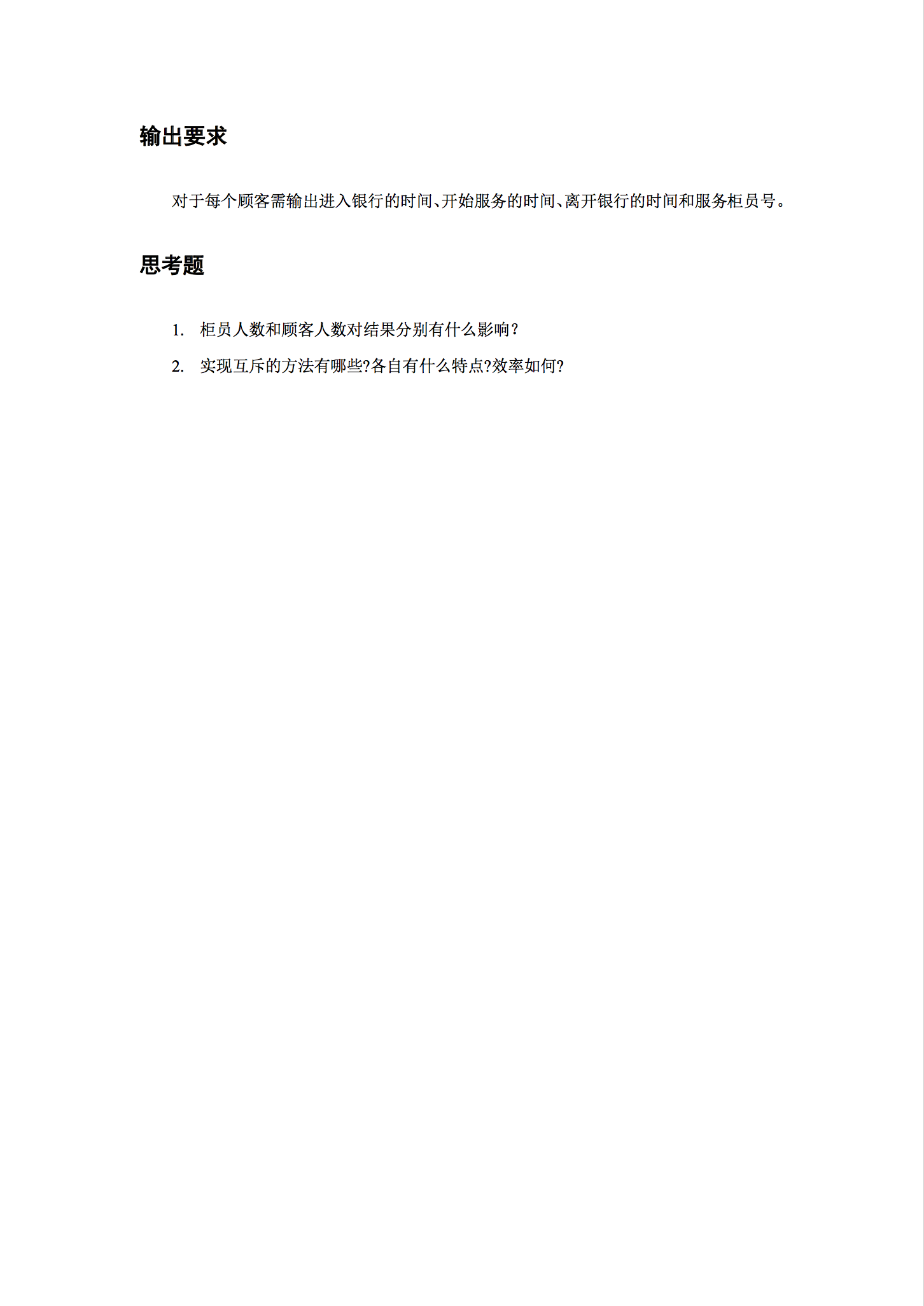
班级：无53

姓名：陈相宁

学号：2015011033

Email：[c.xiangning1997@gmail.com](mailto:c.xiangning1997@gmail.com)



****

1. **实验思路：**

将每个用户（Customer）以及银行柜员（Clerk）当作独立的线程。因此实验关键在于实现线程之间的同步与互斥。

1. 同步：

利用Ticket自建类实现。有customer取号时自动生成一个Ticket对象。其中设置一个初值为0的Semaphore用于判定是否有clerk叫到了该号：如果叫到，则执行V操作，用户停止等待；否则，用户线程阻塞，继续等待。

1. 互斥：

互斥主要防止同时有customer取号和clerk叫号的情况。因此只需设置初值为1的Semaphore mutex即可。

实现了线程的同步及互斥后，还需要一个仓库来生成、存储ticket，为customer发号并分配给clerk任务。

1. **代码实现：**

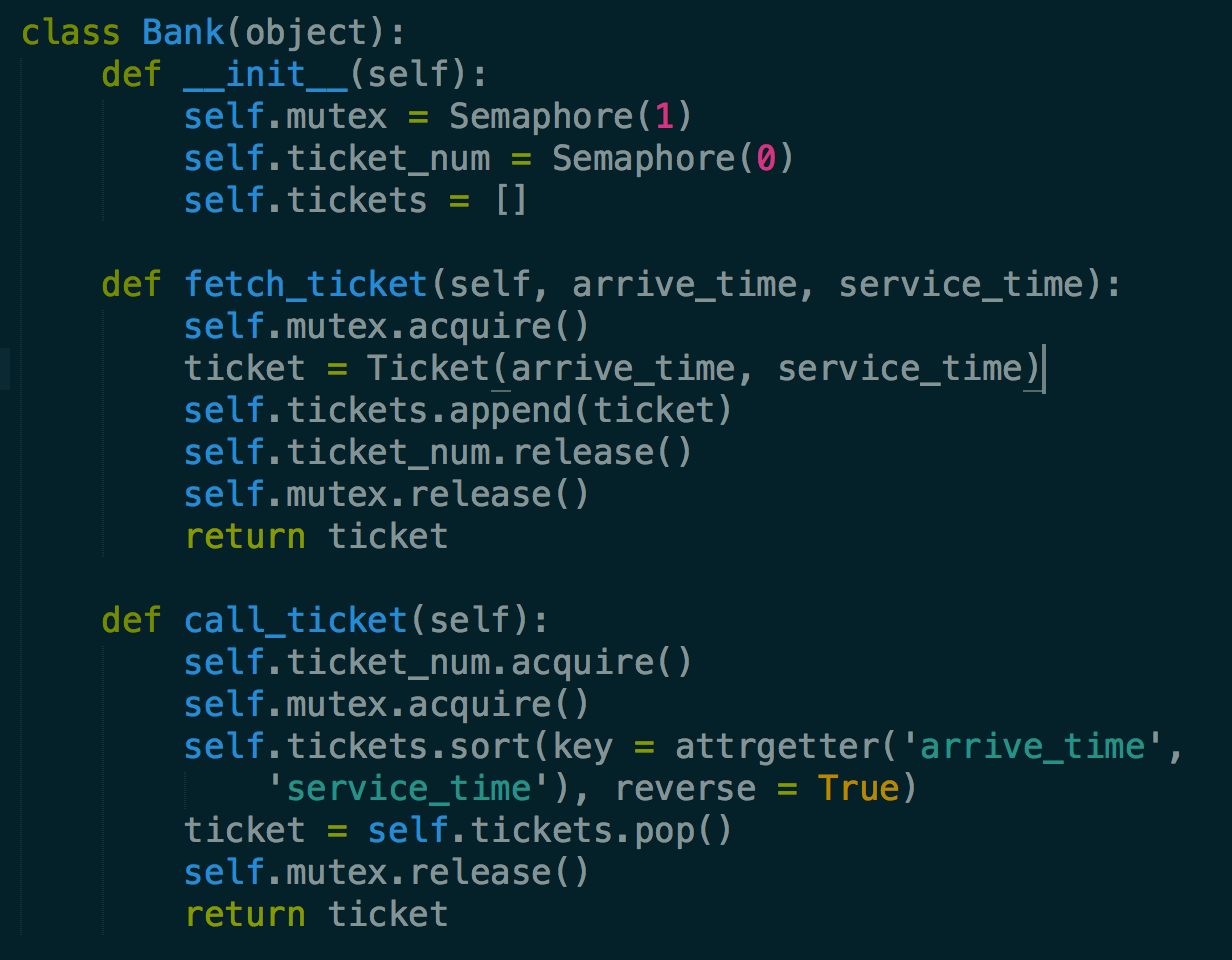
编程语言为python。利用threading库中的Semaphore类实现信号量。Semaphorea.acquire()为P操作，Semaphore.release()为V操作。继承Thread类实现Customer和Clerk。利用time库中的time方法获取当前时间，sleep方法在需要的时候将线程休眠。

1. Bank类：

自建Bank类存储生成的ticket。初始化时建立初值为1的mutex信号量，初值为0的ticket\_num信号量，名为tickets的list来动态存储号。

Bank类包含用户fetch\_ticket以及银行柜员call\_ticket方法。fetch\_ticket方法中，首先对mutex执行P操作保证同时只有一个人访问Bank，然后生成一个Ticket对象并加入tickets list中，同时对ticket\_num执行V操作，最后执行mutex V操作完成一次取票。

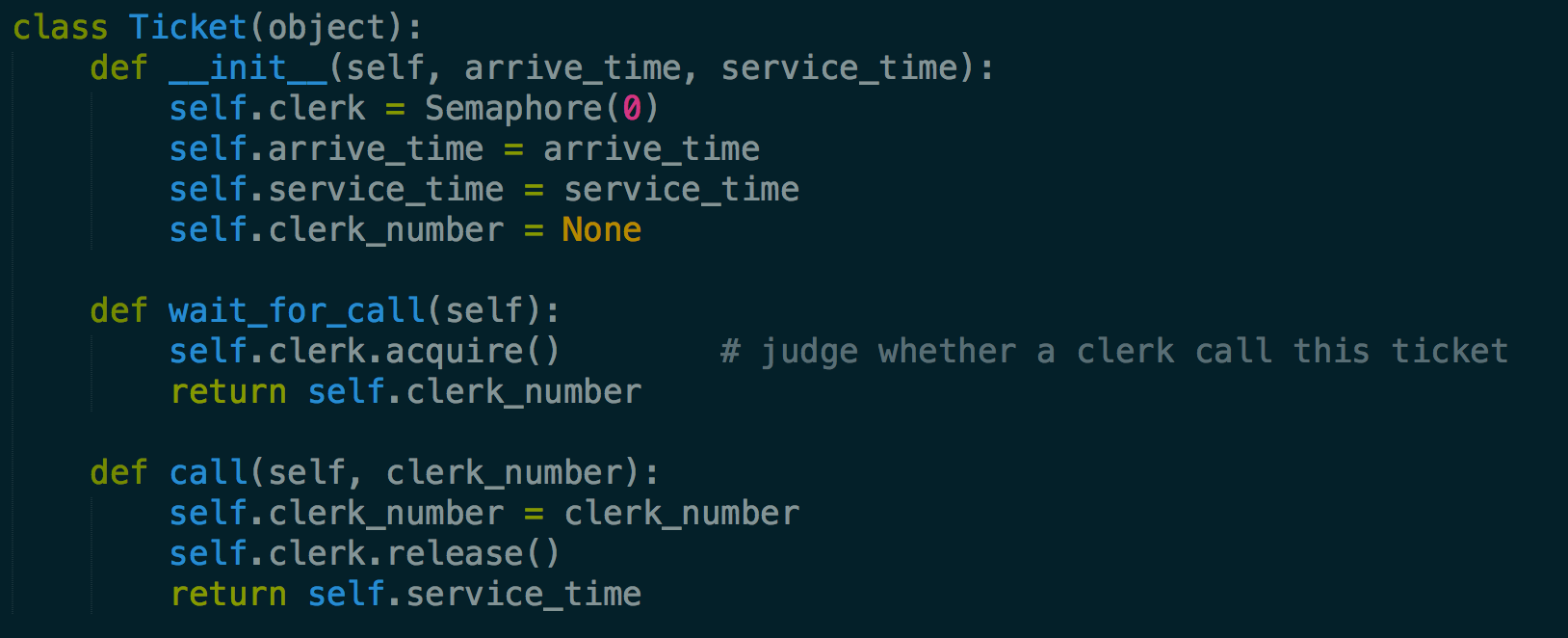
call\_ticket方法中，首先对ticket\_num执行P操作保证仍有customer未被服务，之后执行mutex P操作。之后一步很关键：有两种机制，一种是保证平均服务时间最短，即取tickets list中服务时间最短的号；另一种是保证先来后到的情况下尽量缩短总服务时间，即取tickets list中最先来的一批customer中服务时间最短的。由于后一种较符合常理，因此之后采取第二种叫号方式。最后对mutex执行V操作完成一次叫号。



1. Ticket类：

为了实现Bank类中call\_ticket方法，Ticket对象初始化时需输入取票人编号，取票人到来时间以及服务时间。同时设置初值为0的clerk信号量判断该号是否被clerk叫到。

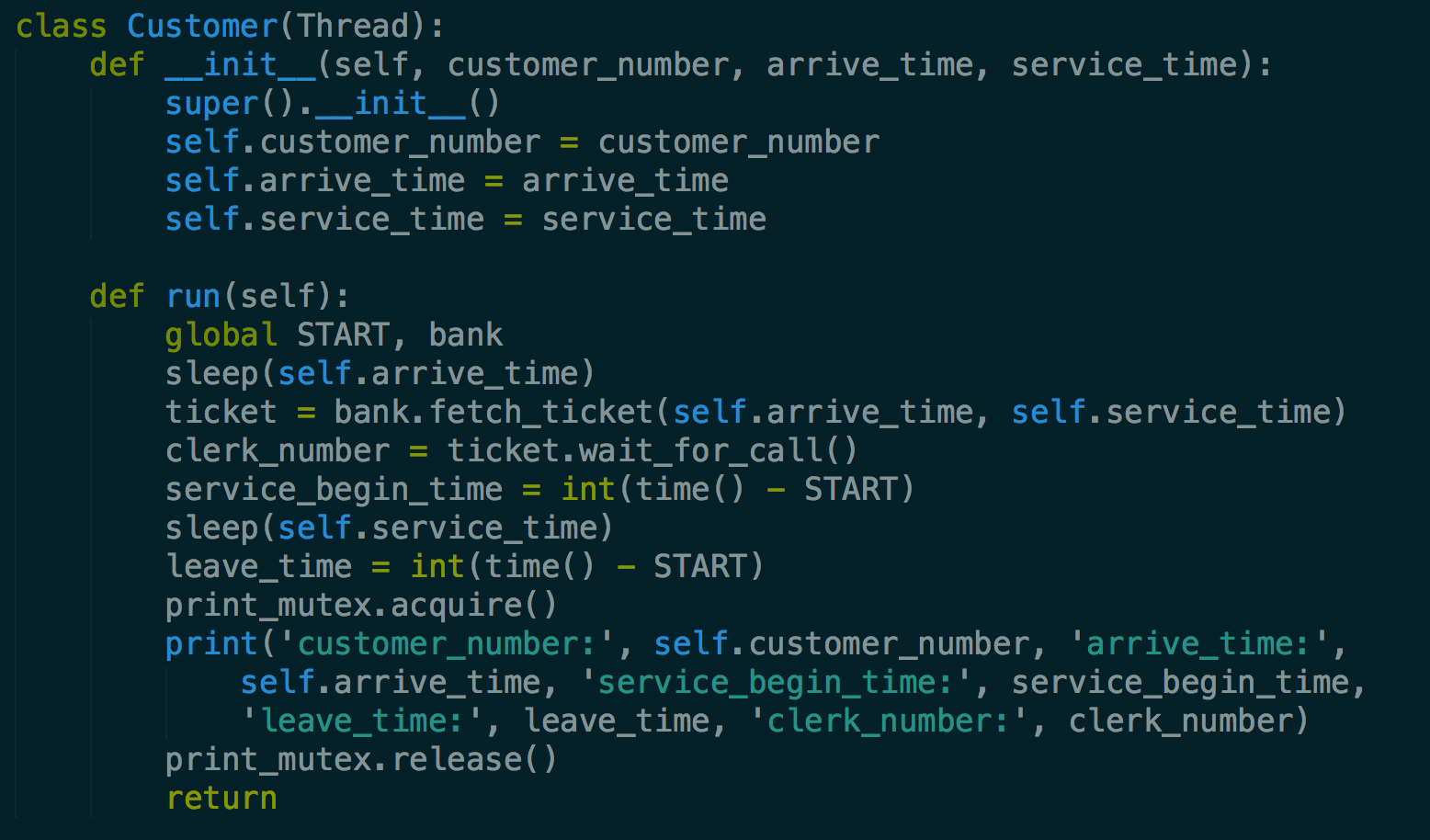
Ticket类中包含用户wait\_for\_call以及call方法。wait\_for\_call方法中，首先对clerk执行P操作，如果有clerk叫号则返回该clerk的编号。call方法中，首先更新该ticket的clerk编号，之后执行clerk V操作唤醒等待中的customer线程，函数返回服务时间以对当前clerk线程休眠。



1. Customer线程：

继承Thread类，初始化时输入customer编号、到达时间以及服务时间。

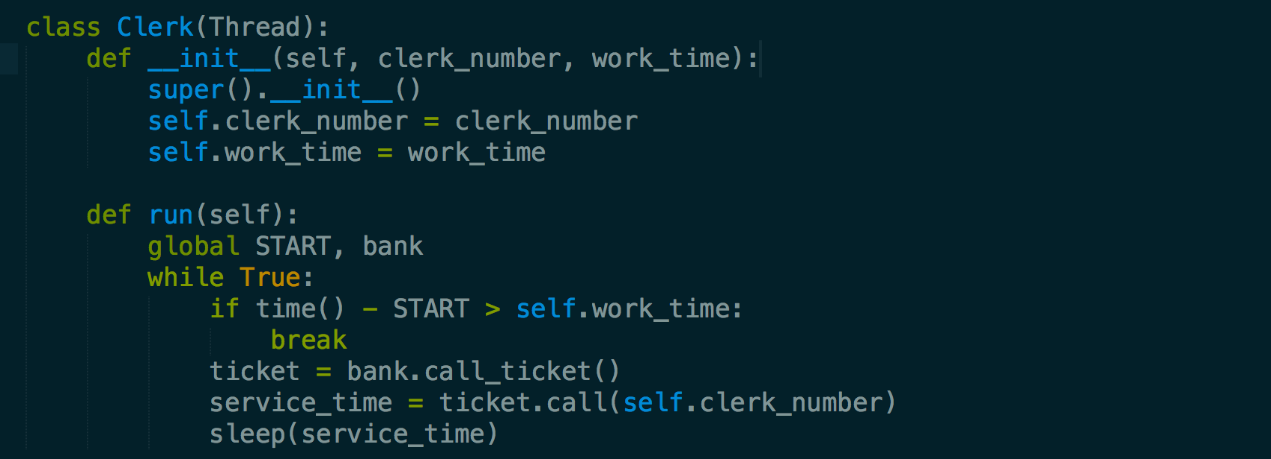
run方法中，先sleep一段时间保证按时到达，到达后调用fetch\_ticket方法从全局Bank对象中取ticket。之后开始等待，某一clerk叫号该ticket唤醒当前customer线程后，再sleep一定时间表示正在服务中，最后线程退出并输出结果。



1. Clerk线程：

继承Thread类，初始化时输入clerk编号，最大工作时间（最大工作时间保证该线程工作一定时间后能够退出）。

run方法中，执行一无限循环，循环中先调用call\_ticket方法从全局Bank中取ticket，取到ticket后对该ticket执行call操作唤醒持有该ticket的customer并获得service\_time，然后sleep相应时间完成一次service。



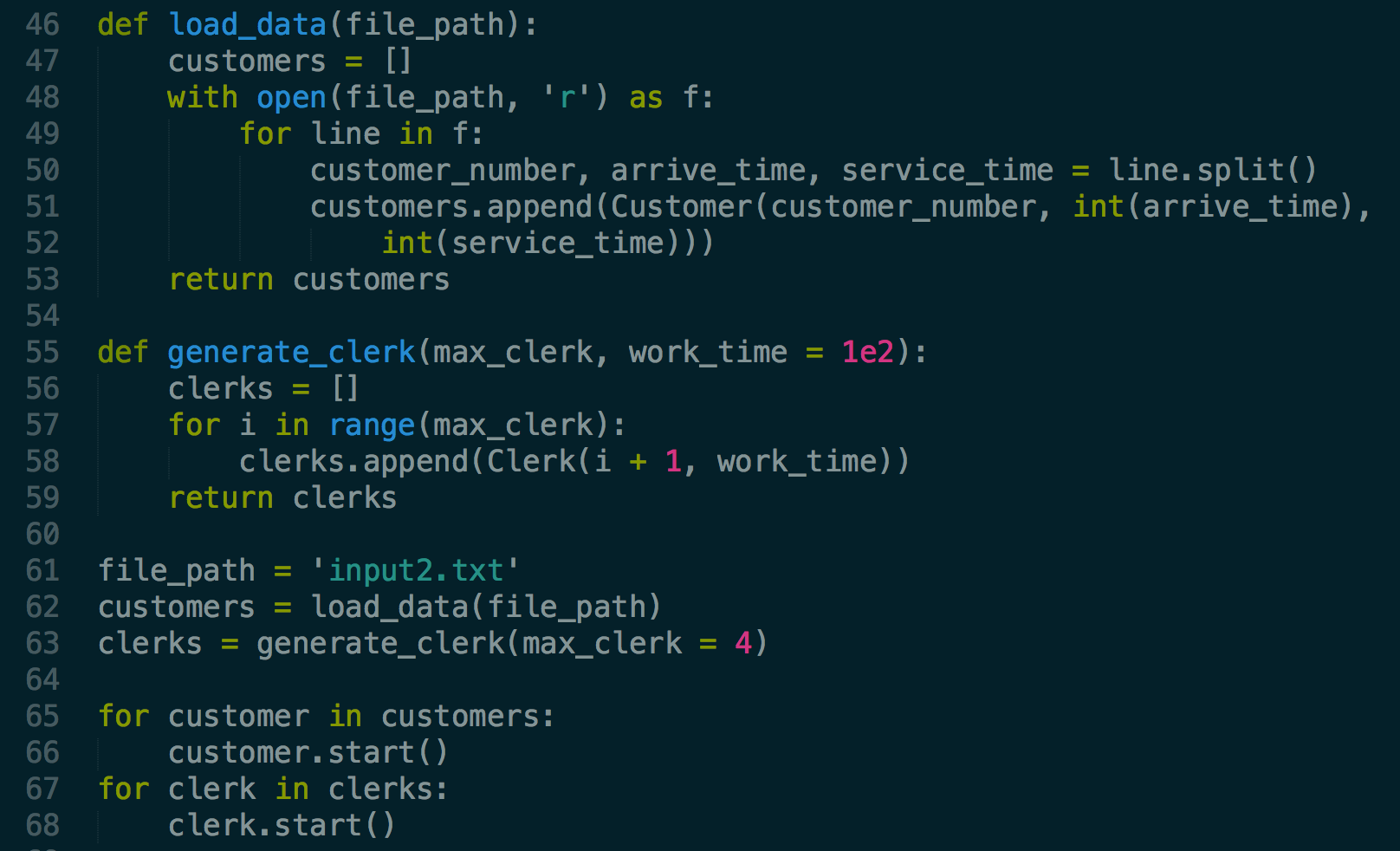
全部代码：

Bank.py：



People\_Thread.py：





测试结果：

1. 简单例子：

输入：

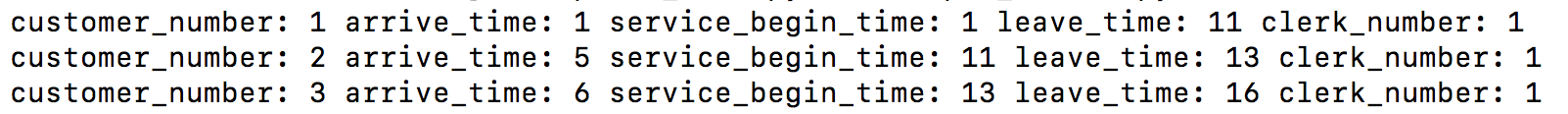
1 1 10

2 5 2

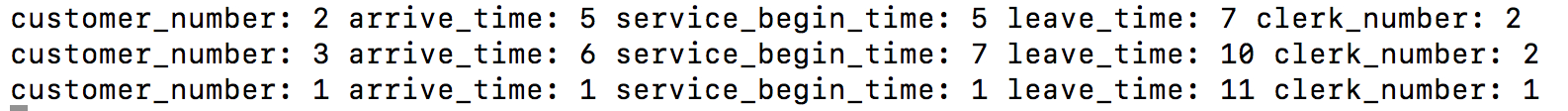
3 6 3

输出：

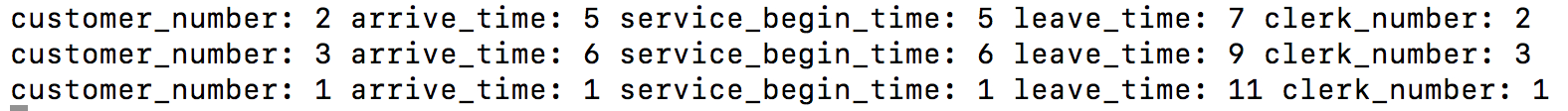
1个clerk：



2个clerk：



5个clerk：



如果只有1个clerk，则总服务时间为10 + 2 + 3 = 15。

如果有2个clerk，则1号clerk服务1号customer，11时结束。2号clerk先服务2号clerk，7时结束。之后服务3号customer，10时结束。

如果有3个以上clerk，则每个customer都能立即被服务。

综上，程序通过小数据测试。

1. 复杂例子：

输入：

1 1 10

2 5 2

3 6 3

4 6 5

5 3 8

6 7 1

7 10 5

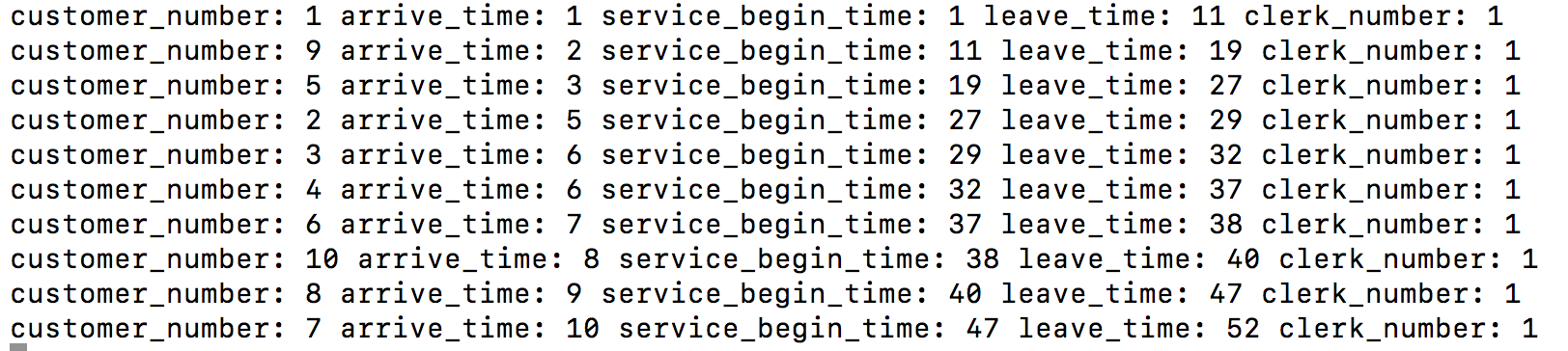
8 9 7

9 2 8

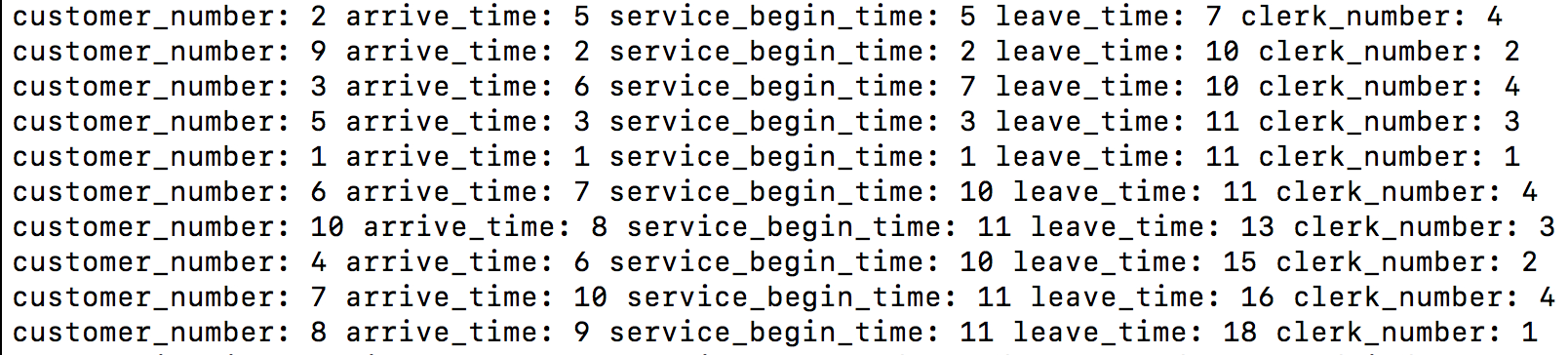
10 8 2

输出:

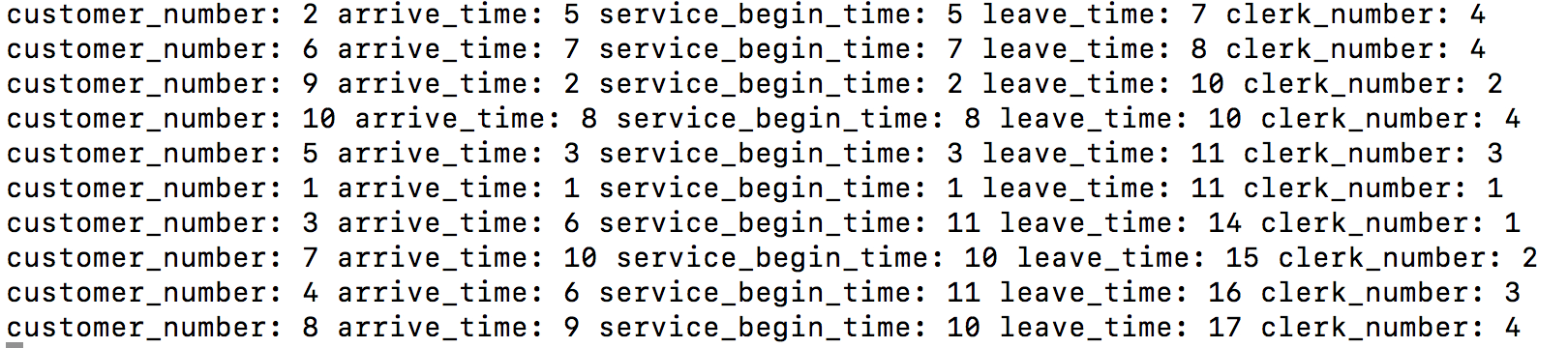
1个clerk：



4个clerk（先到先服务）：



4个clerk（平均服务时间最短）：

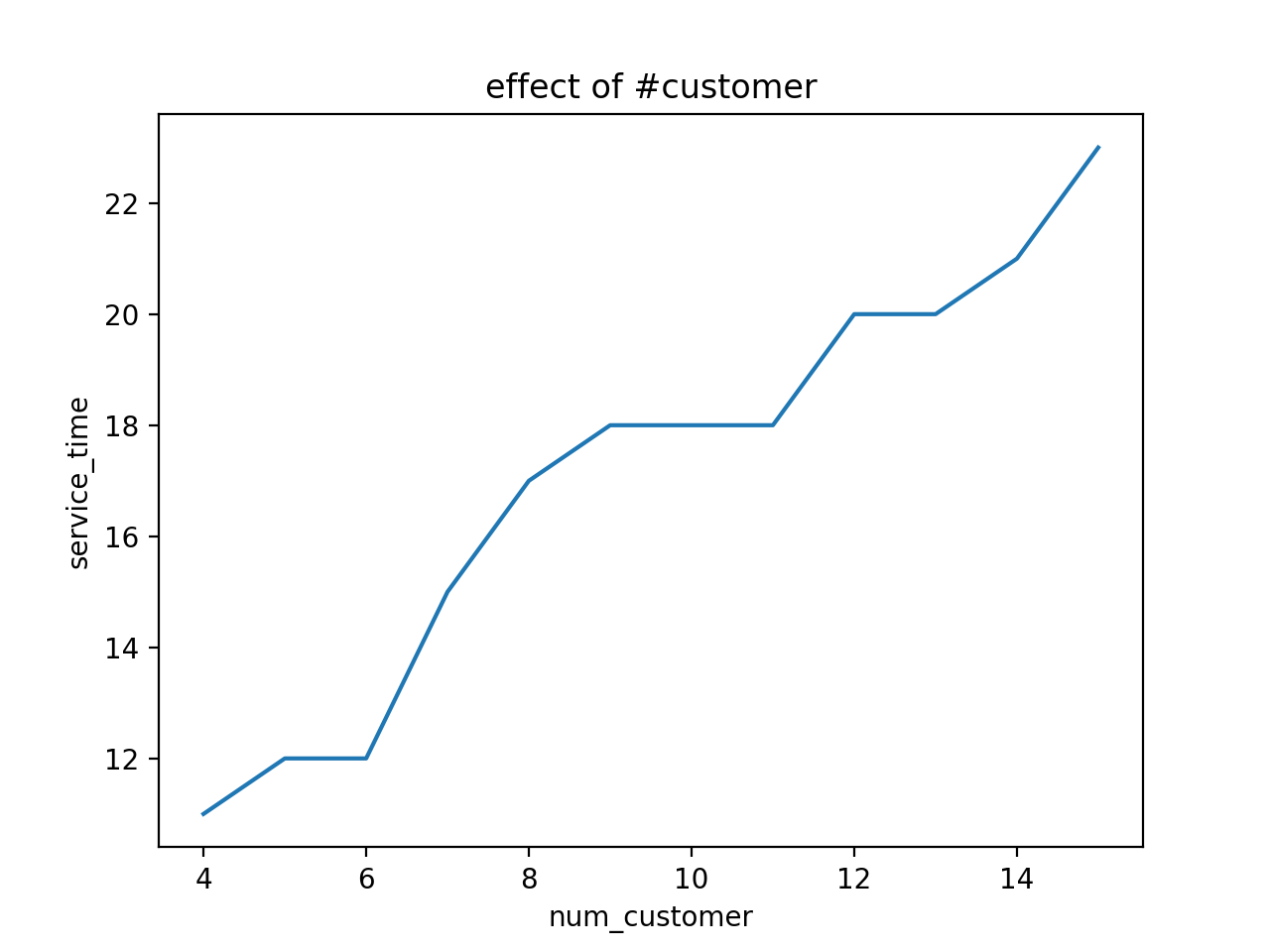


可以看出，该多线程程序运行结果符合预期。先到先服务以及平均服务时间最短方式各有其合理性，应视具体情况而定。（之后测试采取先到先服务模式）

1. **思考题：**
2. **柜员人数和顾客人数对结果分别有什么影响：**

①顾客人数影响：

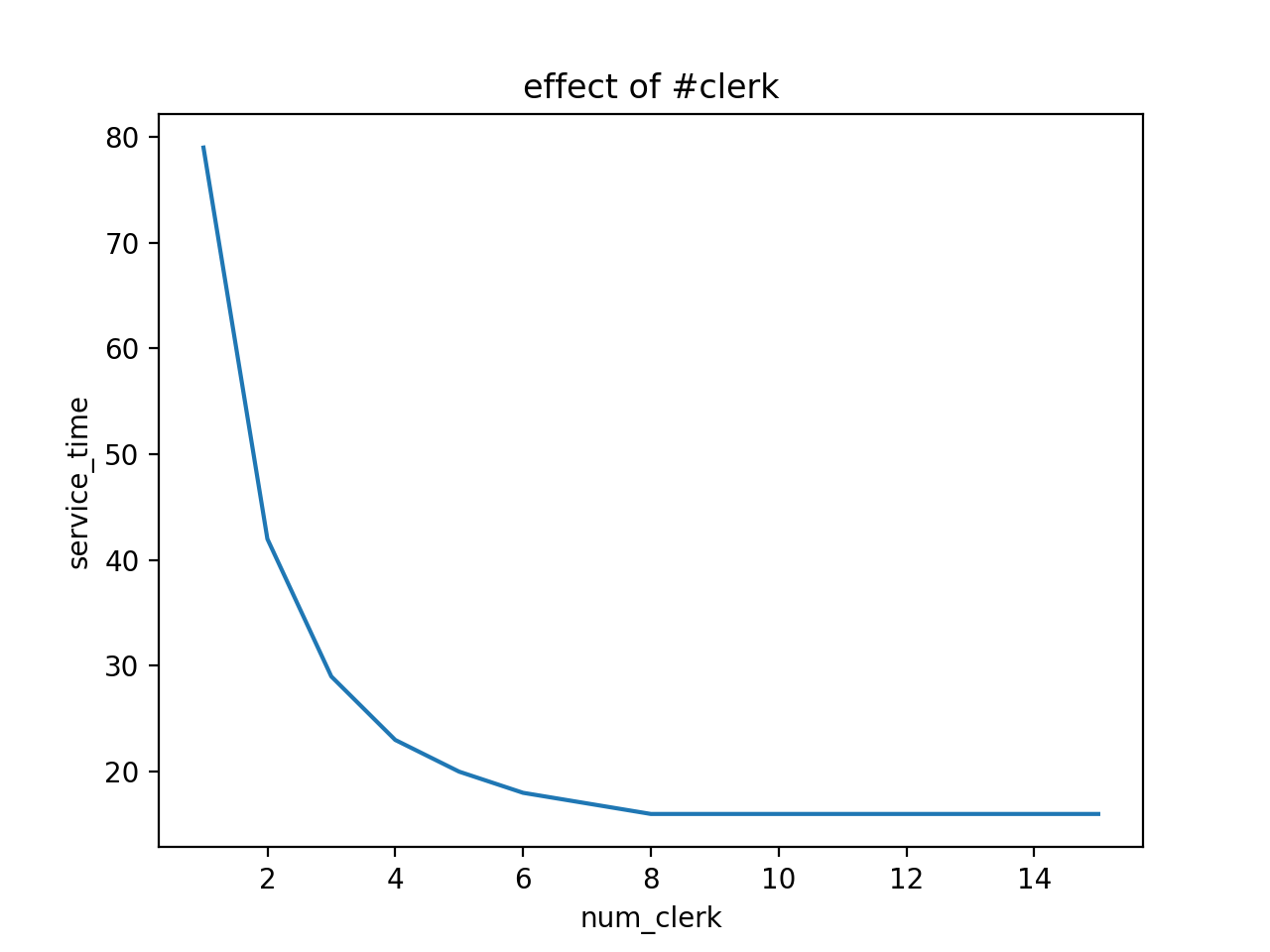
固定柜员人数为4。



结合图像发现，总服务时间随customer数量上升近似线性增长。但具体时间跟每个用户到达时间跟服务时间有关，较为复杂。

②柜台人数影响：

固定顾客人数为10。



结合图像发现，总服务时间一开始随柜台人数增加而减少，但减缓速度越来越慢，最终趋于稳定值。此时，再增加柜员人数对总服务时间无提升。曲线形状类似负指数函数。

1. **实现互斥的方法有哪些？各自有什么特点？效率如何？**

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 特点 | 种类 | 效率 |
| 禁止中断 | 简单。但是把中断权力交给用户导致可靠性较差。且不适用于多处理器 | 禁止中断 | 高 |
| 自旋锁 | 忙等待，浪费CPU时间，只有等待时间非常短时使用 | 忙等待 | 较低，且可能带来优先级反转问题 |
| 严格轮转法 | 要求两个进程严格轮流进入临界区。忙等待问题 |
| Peterson算法 | 严格轮转法的优化，可以正常工作。但仍存在忙等待问题 |
| 硬件指令方法 | 适用于任意数目进程。简单，容易验证正确性可支持进程中存在多个临界区。但仍有忙等待问题 |
| 信号量 | 可实现进程同步，但信号量的控制分布在整个程序中，很难分析其正确性：同步分散操作、易读性差、不利于修改和维护、正确性难以保证 | 信号量 | 较高 |
| 管程 | 信号量的优化，提高代码可读性，便于修改和维护，正确性易于保证。模块化、抽象数据类型、信息封装。但C及多数语言不支持管程 | 管程 | 较高 |
| 消息传递 | 适用于不同机器之间的进程通信 | 消息传递 | 较高 |